**Lecture 24 - Chapter 10: C Structures, Unions, Bit Manipulation – Mon Nov 13 or Tues Nov 14**

**Announcements**

Reading:

* Chapter 10

Assignments:

* Assign: Assignment #10 - due on **Nov 20** (MW class) or **Nov 21** (TR class) **(no late assignments accepted)**

**Today’s Goals**

1. Structures
2. Initializing Structures
3. Accessing Structures Members

**Today’s Terminology**

**Terminology**

* Derived Data Type
  + Created from one or more types of basic data types
  + Most common are pointers, arrays, structures and unions
* Structures
  + Collection of related variables under one name
  + Derived data type
* Self-referential structure
  + When a structure contains as a member that is a pointer to its same structure type
* Union
  + Collection of related variables under one name, but only one at a time contains a value
  + Derived data type
* Bitwise Operators
  + & - AND
  + | - OR
  + ^ - Exclusive OR
* Bit Fields
  + Number of bits in which an unsigned or signed integral member of a structure or union is stored
* Enumeration
  + Set of integer constants represented by identifiers

**Structures**

**Purpose**

* So far, we’ve seen that an **array** is a collection of elements of the **same** type
* A **structur**e will allow us to create a collection of elements of **different** types

**Why Use Structures**

* To define a record to be stored in a file
* To create complex data structures
  + Combining pointers and structures allows us to create data structures like linked lists, queues, etc

**Defining Structures**

* When defining a structure, you are defining
  + The **structure tag** that is name of the structure
  + The **members** the structure will hold
* General form for defining a structure:

Keyword to define structures

**struct structureName {**

**datatype variable;** Structure tag **-** name of the structure

**datatype variable;**

**datatype variable;** Members **-** different values stored in the structure

**};** (int, float, array, even another structure)

**Example**

* Define a structure that holds the information for an employee

**Employee** => Structure tag

Names the structure definition

**struct** employee { Used to declare variables.

**char** firstName[20];

**char** lastName[20]; **Members**

**unsigned** **int** age; firstName, lastName, age, etc

**char** gender;

**double** hourlySalary; **NOTES**:

**struct** employee assistant; Members must be unique in **same** str

**struct** employee \*assistant; Can include primitive, array, etc.

};

**Cannot** contain **instance** of itself!

**Can** contain a **pointer** to its type!

**Derived Data Type** How we build linked list (chap 12)

Constructed from one or more types

Semicolon – ends the structure

*Made of what I specifically need* to solve my problem

This definition **did not** reserve any memory

This definition **did** create a NEW data type – we can now define variables of this NEW type!

* What does this look like?

Structure name

**employee**

firstName … …

lastName … Different members

age 40

gender ‘m’

hourlySalary 50.00

**Including Structures in Code**

* We’ve defined primitive types and arrays in code like this:

**int** **main**(**void**) {

**int** numCars = 15;

**int** numberList[10];

**}**

* Structures are ***generally*** defined before main => global scope
* gives functions ability to use definition

**struct** employee {

**char** firstName[20]; This is **NOT** a global variable!

**… This is a new global type**

**double** hourlySalary;

};

**int** **main**(**void**) { Define BEFORE main if functions need definition

} Define IN main if want scope limited to main

**Defining Variables of Structure Types**

**struct** employee aEmployee; Declares variable to be a structure

Declare with employee & **struct keyword**

**struct** employee employees[100]; Declares array with 100 elements of type

**struct employee**

**struct** employee \*employpeePtr; Declares a pointer to the **struct employee**

Like **int \* employeePtr**

Here the data type is “struct employee”

**Here is where we reserved memory!**

* Could also write above into one strange looking definition:

**struct** employee aEmployee, employees[100], \*employeePtr;

* The above code reserves memory for
  + One **struct employee** object called **aEmployee**
  + 100 **struct employee** objects in the array **employees**
  + 1 pointer to a struct employee in **employeePtr**

**Structure Default Values**

* Like other types, when a ***structure*** is created, it is filled with garbage values NOT zero!

**struct** employee defaultEmployee;

// Showing that a structure is filled with garbage

**printf** ("Employee's first name: %s\n", defaultEmployee.firstName);

**printf** ("Employee's last name: %s\n", defaultEmployee.lastName);

**printf** ("Employee's age = %d\n", defaultEmployee.age);

**printf** ("Employee's gender = %c\n", defaultEmployee.gender);

**printf** ("Employee's hourly salary = %3.2f\n", defaultEmployee.hourlySalary);

**Displays (will vary on each system)**

Employee's first name: ¨þ(

Employee's last name:

Employee's age = 196608

Employee's gender =

Employee's hourly salary = -0.00

* Should always initialize structures with an appropriate default value. Here is one way to do so:

**struct** employee defaultEmployee;

// Showing one way to initialize structure

**strncpy** (defaultEmployee.firstName, " ", NAME\_SIZE);

**strncpy** (defaultEmployee.lastName, " ", NAME\_SIZE);

defaultEmployee.age = 0;

defaultEmployee.gender = ' ';

defaultEmployee.hourlySalary = 0;

**Structure Initializers**

* An easier and faster way to initialize arrays and structures is with initializer lists
* Used initializer lists with arrays

**int** numberList[5] = {3, 34, 44, 81, 7};

* Use with structures as well

**struct** employee aEmployee = { "Joe", "Smith", 23, 'm', 15.00 };

* If **fewer** initializers, remaining members are initialized to 0 (or NULL).

**struct** employee aEmployee2 = { "Joe", "Smith" };

* If **more**, it gave warning and ignored! (Did not overwrite!)

**Note:**

* Can I initialize values inside the definition?
  + Cannot do the following

**struct** employee {

**char** firstName[20] = {" "}; **Error!** Cannot do this.

**char** lastName[20] = {" "}; No memory has been allocated

**unsigned** **int** age = 0;

**char** gender;

**double** hourlySalary = 0.0; **Error**! – cannot contain

**struct** assistant employee; instance of itself

};

* Valid operations on structs
  + Assigning one struct to another struct of the **same type**
  + Taking the address (&) of a struct variable
  + Assessing members of a struct variable
  + Using the sizeof operator on a struct variable
* Invalid operations on structs
  + Comparing **one struct** to **another struct**
    - **Can** compare members

**if** (employee1.age == employee2.age) {

**puts** ("Comparing structure members is allowed");

}

* + - **Cannot** compare entire struct = > struct members might not be in ***consecutive memory***

**if** (employee1 == employee2) {

**puts** ("Comparing entire structures is not allowed");

}

Reason you can’t do this is because of:

* Memory alignment consideration
* There could be holes in a structure
* Holes are created when system stores memory on different boundaries

**Accessing Structure Members 10 min**

**Accessing Structure Members**

* To access the individual members within a structure use
  + Structure member operator – dot operator (.) and
  + Structure pointer operator – arrow operator (->)

**Example**

* Using the dot operator to print information in **aEmployee**

**printf** ("Employee's first name: %s\n", aEmployee.firstName);

**printf** ("Employee's last name: %s\n", aEmployee.lastName);

**Displays**

Employee's first name: Joe

Employee's last name: Smith

* Using the arrow operator to print information pointed to by **employeePtr**
  + Create using the dash (-) and greater than (>)
  + Used to access members via a pointer to the structure

**struct** employee \*employeePtr; // employeePtr will point to an employee type

employeePtr = &aEmployee; // employeePtr given address of aEmployee

|  |  |  |
| --- | --- | --- |
| aEmployee | **0028FF10** |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| employeePtr |  | **0028FF10** |

**printf** ("Employee's first name: %s\n", employeePtr->firstName);

**printf** ("Employee's last name: %s\n", employeePtr->lastName);

**printf** ("Employee's age = %d\n", employeePtr->age);

**printf** ("Employee's gender = %c\n", employeePtr->gender);

**printf** ("Employee's hourly salary = %3.2f\n", employeePtr->hourlySalary);

* Now we can see how the initializer works

**struct** employee aEmployee = { "Joe", "Smith" };

**puts** ("Example of using an initializer list with a struct");

**printf** ("Employee's first name: %s\n", aEmployee.firstName);

**printf** ("Employee's last name: %s\n", aEmployee.lastName);

**printf** ("Employee's age = %d\n", aEmployee.age);

**printf** ("Employee's gender = %c\n", aEmployee.gender);

**printf** ("Employee's hourly salary = %3.2f\n", aEmployee.hourlySalary);

**Displays**

Employee's first name: Joe

Employee's last name: Smith

Employee's age = 0

Employee's gender =

Employee's hourly salary = 0.00

* Note: employeePtr->firstName
  + Same as **(\*employeePtr).firstName**

Dereference pointer -> gets us to the structure **aEmployee**

Dot operator -> gets the member value of firstName

**printf** ("First name: %s\n", (\*employeePtr).firstName);

**printf** ("Age = %d\n", (\*employeePtr).age);

**Not including parentheses results in an error!**

**printf** ("First name: %s\n", \*employeePtr.firstName);

**Why?** Dot operator has higher precedence and dereference operator

**Note**

* Good programming practice
  + **DO NOT** put spaces around the dot or arrow operator
  + This still works

**printf** ("employee1 . firstName = %s\n", employee1 . firstName);

**printf** ("employeePtr -> firstName = %s\n", employeePtr -> firstName);

* + Omitting spaces emphasize it’s a single variable name

**Examples – referring elements**

* + Different ways to reference the first member (array - firstName)

**printf** ("&employee1 = **%p**\n", &employee1);

**printf** ("employee1.firstName = %p\n", employee1.firstName);

**printf** ("&employee1.firstName[0] = %p\n", &employee1.firstName[0]);

**printf** ("employeePtr = %p\n", employeePtr);

Displaying addresses

|  |  |  |
| --- | --- | --- |
| aEmployee | **0028FF10** |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| employeePtr |  | **0028FF10** |

**Displays**

&employee1 = 0028FF10

employee1.firstName = 0028FF10

&employee1.firstName[0] = 0028FF10

employeePtr = 0028FF10

**printf** ("employee1.firstName = %s\n", employee1.firstName);

**printf** ("employee1.firstName[0] = %s\n", employee1.firstName[0]);

**printf** ("employeePtr->firstName = %s\n", employeePtr->firstName);

**printf** ("(\*employeePtr).firstName = %s\n", (\*employeePtr).firstName);

**printf** ("\*employeePtr.firstName = %s\n", \*employeePtr.firstName); **Error!!**

**Displays**

employee1.firstName = Joe

employee1.firstName[0] = Joe

employeePtr->firstName = Joe

(\*employeePtr).firstName = Joe